TerraWeather
Ensuring Your Safety & Success

AI Workshop – Lecture 4

Training & Evaluating Neural Networks

4.0 Good Generalization in Time-series Predictions

A successfully trained NN is one that performs well on the test dataset (this nearly always implies decent performance on the training set.) After all, that is what we are after – a network that can successfully predict the labels of novel input. A network that performs well on the test dataset is said to have *good generalization*.

For categorization tasks (like image classification) successful risk minimization on the training set is closely related to good generalization. This is because the input data are randomly sampled, effectively making the test dataset probabilistically “similar” to the training dataset. The data within training/test sets are independent and identically distributed (i.i.d).

This is not true for time-series problems:

1. **Time-series inputs are not randomly sampled**, because the time ordering of data points matters. (We can chop up a time-series into “mini-batches” but this causes problems, which we discuss in Lectures 6 and 10).
2. **Time-series are often non-stationary.** This means that the training dataset (earlier in time) may be qualitatively different from the test dataset (later in time).
3. **We are often interested in rare events**. This means that even with a large dataset, the number of events we want to predict (eg, dengue outbreaks, equipment failure, algal blooms) are rare or occur infrequently. If they were frequently occurring, predicting them wouldn't be of much value. This rarity often precludes training/testing sampling based on events. Instead the best option is to divide the time-series into training/testing periods based on a single point in time, making sure that the test period contains some of the events we are interested in predicting. However, it implies that the events in training vs. test datasets aren't i.i.d.

For these reasons, when it comes to time-series problems, it is neither obvious or certain that risk minimization (through **any** algorithm) will give you a network that generalizes well. Risk minimization only guarantees successful learning of the training dataset.

Instead, you must first check that conditions (2) and (3) aren't occurring in your datasets. This can be done through visual inspection: plot the labels and look at the training/test datasets. Are they visually similar? Do the same for the inputs. If they are very different, you may run get poor generalization.

4.1 Aggregated Labels

In our work at Terra Weather, we've found it useful to pay attention to the way a label is created. *Aggregated labels* are those that are either implicitly or explicitly created by aggregating the output of many contributing processes.

The *aggregation method* could be summing, averaging (or using the median) or extremal values (maximum/minimum). Figure 4.0 illustrates:

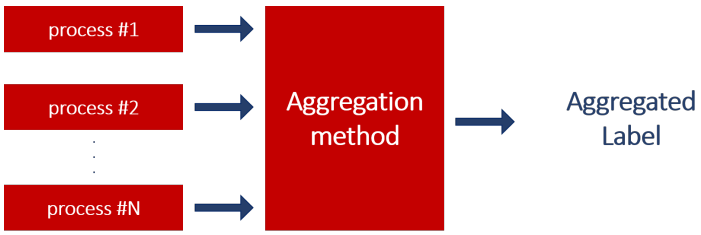

Figure 4.0 – Aggregated label from N contributing processes.

For example, predicting the “maximum temperature” over Singapore would involve **explicit** aggregation since the temperature readings from many sensors have to be aggregated (using “maximum” as aggregation method) to obtain a “maximum temperature” label for training. Explicit aggregation means you have the output of each contributing process and explicitly run the aggregation method yourself.

As a second example, predicting the “number of dengue cases” in Singapore for a given week would involve **implicit** aggregation since these cases come from all over Singapore, and contracting dengue is a relatively local phenomenon (you need to be bitten by a mosquito, which don't fly very far from where they spawn). So, the weekly dengue case number is an aggregated label involving numerous spawn sites all over the country. Implicit aggregation happens when you don't know the number of contributing processes or their output (ie, dengue cases caused by these sites); the aggregation is implicit in the problem.

Aggregated labels are problematic because:

- **they can amplify outliers** – in the case of maximum or minimum, or,
- **they may cause some or all of the time-series to be impossible to predict** – in the case of implicitly aggregated labels, a combination of variability and non-stationarity in the contributing processes and

missing input data (you don't always have the sensors in the right place at the right time).

Some ways to “solve” these problems? Consider:

- **Running separately trained NNs** to predict each contributing process separately, (in the case of explicit aggregation). These results can then either be naively aggregated or be aggregated through another NN specially trained for that purpose.

- **Retaining as much as possible of the inputs**, without aggregating them. For example, if you have 5 stations, use them all as input, without aggregation (eg averaging). The issue here is dealing with missing values. Unfortunately, there is no magic bullet for this and each method has its drawback.

4.2 How Good is Good? Using Benchmarks

One early issue you need to deal with is objectively comparing the performance of your NNs. The training and test losses are useful, but they only measure improvements in performance. They don't tell you how your “good” your predictions are. For this you should always compare your predictions against a *benchmark*.

A widely used benchmark is *persistence*. Persistence says that for a label $f(t+\Delta)$ where Δ is the time-horizon, the “prediction” $\hat{f}$ is $\hat{f}(t+\Delta)=f(t)$

Persistence is a simple, universal benchmark, and we suggest you use it in your work. Plot it on your time-series graphs and normalize your losses against persistence. For example, if on your test dataset the loss is 2.9 and the loss from a persistence prediction is 2.8, you should tabulate the normalized loss as

$$\frac{NN\ loss}{benchmark\ loss}=\frac{2.9}{2.8}=1.04$$

Normalized losses < 1 indicate improvement over the benchmark.

As you improve your networks, you should replace persistence with your best performing network as a benchmark to gauge improvement.

4.3 Overfit

Overfit is a problem that manifests itself as good training performance but poor testing performance (ie, poor generalization).

Note that not all poor generalization is caused by overfit – non-stationarity (discussed in section 4.0) and a lack of prognostic variables (discussed in Lecture 10) can also cause poor generalization. However, overfit is a very common problem during training.

An indicator that overfit is happening is *a sudden change* in prediction quality from training period into test period (see Figure 4.1a). If this behaviour is still manifest when the test set is either substantially decreased or increased, then you are much more confident that overfit has occurred.

Compare Figure 4.1a with Figure 4.1b, which has far less overfit because the performance on the training/test set are very similar. Unfortunately, this is still a bad prediction because it is comparable to persistence (see Figure 4.1c). So, the cause of poor generalization in Figure 4.1b and 4.1c more likely a lack of prognostic variables in the input data.

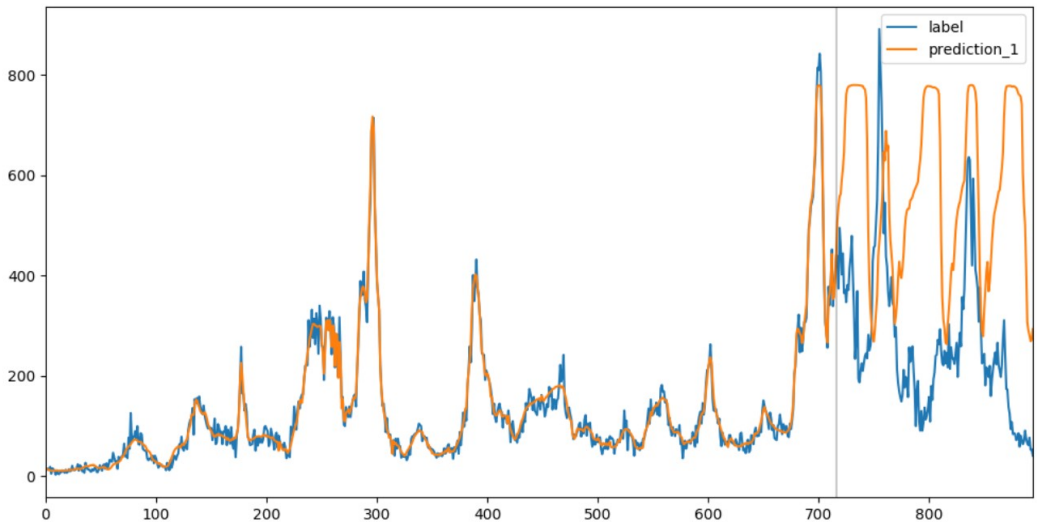

Figure 4.1a – A prediction with overfit. labels (blue) and predictions (orange) over training/test periods. The grey vertical line delimits training/test

periods.

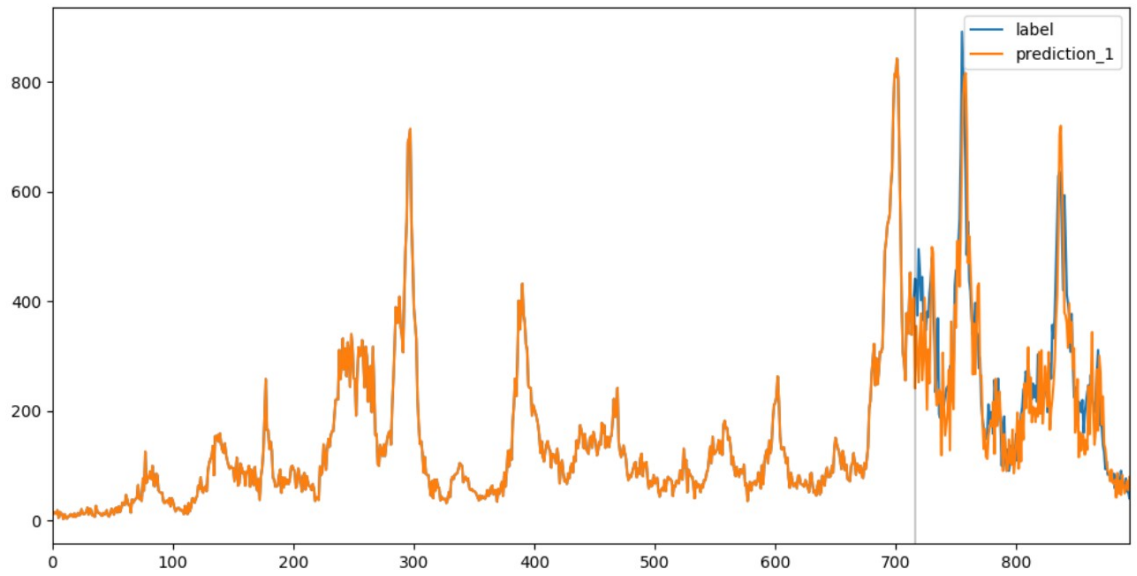

Figure 4.1b – A prediction with very little overfit. labels (blue) and predictions (orange) over training/test periods. The grey vertical line delimits training/test periods.

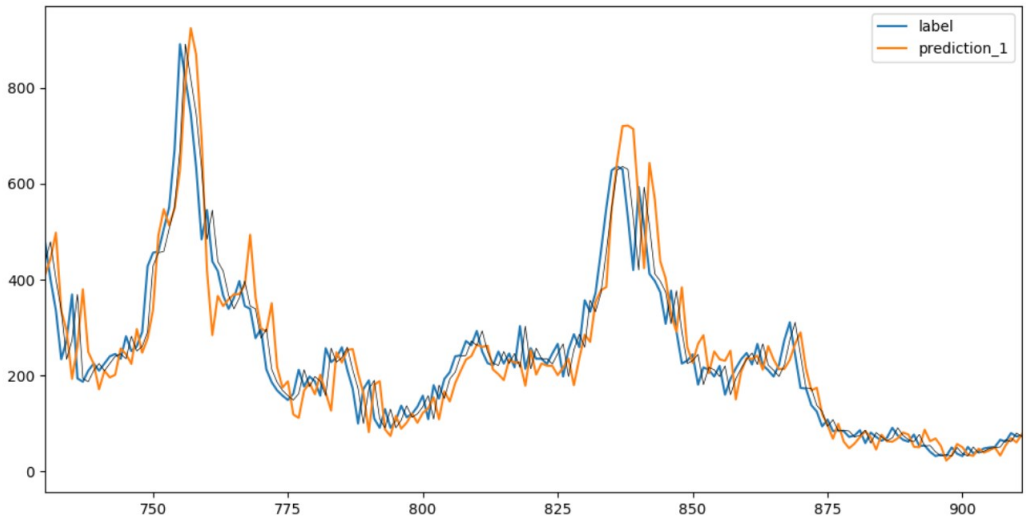

Figure 4.1c – Zoom of Figure 4.1b for test period only. labels (blue), predictions (orange), persistence (grey). The predictions are comparable to persistence. (same input data as Figure 4.1a).

A second (but less reliable) sign of overfit is small training loss but large testing loss. Figure 4.1d shows this behaviour, but be warned that overfit can occur even if the loss graphs look “normal” as in Figure 4.1e.

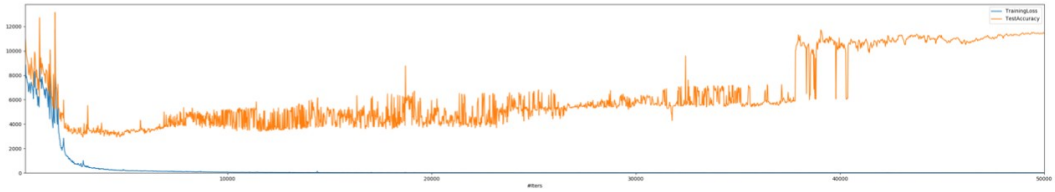

Figure 4.1d – A clear case of overfit in the loss graphs (blue = training, orange = testing) over training-phase iteration.

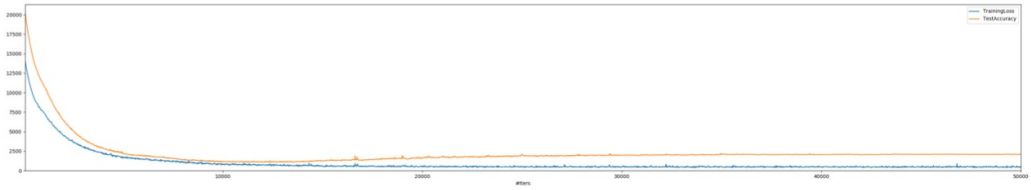

Figure 4.1e – training (blue) and test (orange) losses over training-phase iteration. This is a good candidate for early stopping.

4.4 Curing Overfit

Some ways to cure overfit:

- **Dimensional Reduction**. Neural networks frequently overfit when the input dimensions are too big. We discuss this in Lecture 7.

- **Early Stopping** is a method to stop training if the test losses don't improve after a given number of iterations. This method is not available in standard Caffe, but we have put in a patch for it. Your installation of TW-Caffe should have this patch. Figure 4.1c shows a good candidate for early stopping, since the test loss initially decreases but later increases. This can be prevented by early stopping.

- **Regularization** sets limits to (a) the number of non-zero weights or (b) the total size of weights. Regularization can be a powerful tool, but should be used towards the end of model training. We discuss best practices in Lecture 10. Regularization is similar to LASSO from statistics.

- **Dropouts** randomly remove neurons within your network. From experience, we feel this gives unpredictable results, so use it with care, if at all.

4.5 Start with Overfit

Overfit is not always a bad thing, and in fact, you should aim for overfit initially.

You want to start with overfit because it guarantees learning on the training set. If your network fail here, there's little chance it will fare well on the test set.

Big networks with lots of weights will often overfit. So, you should start work on the smallest network that begins to overfit, only then apply the methods of section 4.3 to gradually improve generalization. You should try different network sizes using Autocaffe's expander system (Lab 3).

4.6 Being Sure of your Results

Two training runs of the *same* NN model will often give quite different predictions, because some settings (the solver, random initializations, etc) are stochastic in nature.

So, to validate your network's architecture and the validity of your approach, you must collect statistics of your NNs by re-training them many times, then analyzing the performance statistics.

The two key statistics to pay attention to are the mean performance and the standard deviation. You want low mean (ie, small losses) and also low performance s.d.

Autocaffe has a “repeat” system that allows you to automatically run many training repeats of the same NN model. Autocaffe also has an `ac stats` command to generate mean and s.d. statistics. You'll use both in Lab 4 shortly.

Besides these statistics, the other key tool to visually evaluate performance of

a large number of models are *Spaghetti diagrams*. Essentially, you overlay the predictions of all your repeats together with the labels, as in Figure 4.2.

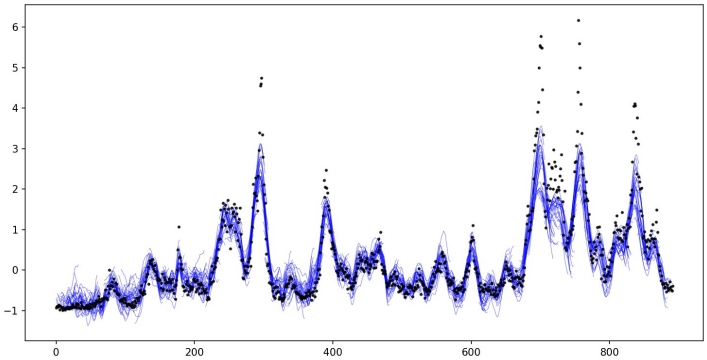

Figure 4.2 – Spaghetti diagrams of 24 repeat models (light blue lines) against labels (black dots)

Spaghetti plots are useful because they show consistent trends in your repeats, allowing you to debug problems in your models.

For example, in Figure 4.2, it is clearly apparent that the models consistently underpredict high peaks in the labels. This could be caused by overly aggressive regularization or too small a network size.

Quiz 4: (You should attempt to answer **all** these quiz questions before proceeding to Lecture 5.)

1. If we had tried to use all the data for training (ie, no test set), then risk minimization would probably tune our model so that it correctly "predicts" the entire time-series. So, why not do this? Why use a test set?

2. Explain clearly how would non-stationarity affects your model's predictions. How would you handle non-stationarity if you can't ensure that it is eliminated from your data?

3. How is explicit aggregation different from implicit aggregation? Why is this distinction important?

4. Why do the maximum/minimum aggregation methods amplify outliers?

5. Would early stopping have helped in Figure 4.1d? Explain your answer.

6. Is it always possible to reduce both average loss and the loss s.d.? Why not?

7. * What are some methods to handle missing values in input? Carefully list their (dis)advantages.

8. * What are 2 reasons why the prediction in Figure 4.1c “bad”? Why did we rule out overfit? Why did we suggest a lack of prognostic inputs rather than non-stationarity as a possible cause of the “bad” prediction?

9. * Why would aggressive regularization or a small network size cause the models in Figure 4.2 to underpredict peaks? Which of the two explanations is more likely? Defend your answer.

10. * Joshua wants to automate the elimination of NNs based on losses alone. Is this a wise decision? Explain your answer clearly.

11. ** Muthu boasts normalized test (euclidean) losses of 0.95 in his latest NN. Does this mean his prediction is “better” than persistence? Carefully defend your answer.

Lab 4: Predicting the mean temperature 1, 4, 8 weeks ahead ("time horizon") with persistence benchmark.

In this lab, you will:

1. Use the autocaffe repeat “task” and gather stats on the repeated experiments
2. Use the persistence model as a benchmark for loss comparison
3. Test the effectiveness of early stopping

Instructions

We will continue to improve the set up in the previous lab, so copy the `lab3` experiment folder to a new `lab4` folder.

Persistence benchmark
Setting up the benchmark of your model against persistence requires changes to the two ways which your models are assessed – the prediction plots and the loss metrics.

Prediction plots

Autocaffe allows you to easily add a plot of the persistence model, however you have follow this convention in **`test.prototxt`**: the name of the blob containing the model prediction must be set as `prediction_n`, where `n` is the prediction horizon of your model.

To make this change, open your `test.prototxt`, go to the final Power layer (the layer named “add”) and rename its top blob, ie change the line

```
top: "prediction"
```

to

```
top: "prediction_8"
```

This will tell autocaffe’s plotting scripts to add a T+8 persistence model when plotting the model predictions.

Report

To print in the pdf report the loss ratios with respect to the persistence model instead of the actual loss, change the two lines in `report.tex_template` from

```
Train loss: $LOSS_TRAIN \\
Test Loss: $LOSS
```

to

```
Train loss: $PLOSS_TRAIN \\
Test Loss: $PLOSS
```

Note: you have to specify a persistence benchmark for the automated stats generation by the repeats task to work.

Repeats

Autocaffe has a “task” system that can schedule and run multiple jobs that belong together in the same experiment (“task”). One such task that can be used is the `repeat` task that duplicates the target experiment a set number of times, runs them all, then compiles the loss statistics.

To run a `repeat` task:

1. Configure the number of repeats to perform by adding this line to `config.txt`

   ```
   REPEAT: 3
   ```

2. Prep the experiment

   ```
   ac prep lab4
   ```

3. Add the tasks

   ```
   ac add-task lab4/* repeat
   ```

4. Wait for the tasks to complete. Check which tasks are still running by using

   ```
   ac tasks
   ```

5. Compile the stats from all the `lab4` tasks

   ```
   ac stats-compile lab4
   ```

6. Generate a pdf report

   ```
   ac report lab4/*/1
   ```

   Note the the plots in the report are taken from the first repeat job from each task. You can change the last number in the above command to use the plots from the other repeats.

To cancel a task:

1. Use the `del-task` command

```
ac del-task lab4/*
```

2. then stop all running jobs and clear the job queue with

```
ac del lab4/*
ac stop lab4/*
```

When a repeat task is run on a directory (eg. `lab4/1`), it reads the `REPEAT` parameter in `config.txt` of that directory and makes the appropriate number of repeat jobs, each in a subfolder (eg `lab4/1/1` to `lab4/1/5` for 5 repeats). It then add those jobs to the queue and waits for them to complete.

Upon job completion, the repeat task runs the `test` and `post` commands on each subfolder, then writes the compiled loss values in the a `stats.txt` file located in its directory (eg `lab4/1`). The `stats-compile` command then reads all the `stats.txt` files in the subfolders and compiles them into a single `stats.txt` file in the parent directory (`lab4/`)

Lag

One of the metrics reported in `stats.txt` is the "lag". Lag gives a measure of how much delay a prediction has from the label. For example, if a model is able to perfectly predict T+1 values, but always predicts it one time-step later (ie it predicts $\hat{f}(t+2)=f(t+1)$), then it has a lag of 1.

Lag values complement the loss in providing a better description of the performance of a model and will be discussed in detail in the next lecture.

Early stopping

As stock Caffe does not have an early stopping feature, Terra Weather has developed an implementation that has been installed in this version of Caffe.

There are 2 parameters to specify

- `length`: maximum number of tries to check the test loss for improvement (ie training stops after no improvements in test loss after `length` attempts)
- `skip`: perform check for early stopping condition once every `skip` iterations

To use early stopping in your experiments, simply add to `solver.prototxt` an extra early stopping parameter, eg

```
early_stop_param {
```

```
        length: 10
        skip: 5
}
```

Quiz
Try out all the techniques described in this lab and see if you can deduce or demonstrate:

Overfit
- How can you identify which experiments are overfitted?
- Which configurations are more prone to overfitting and why?

Early stopping
- Is early stopping effective in preventing overfitting?
- Does the input batch size affect the effectiveness of early stopping?

Persistence benchmark
- Is looking at the visual (predictions plots) or loss ratios more effective in evaluating model performance? What are the advantages and disadvantages of each method?

Repeats
- Does repeating an experiment multiple times really improve the reliability of your evaluation of the models?
- What factors affect the reproducibility of your models?
- How do you decide on the number of repeats per experiment to do?

Homework

To give you a taste of working on a real world problem, we'll try to predict the number of dengue cases per week in Singapore.

Dengue dataset
As described in Section 3.12 of the previous lecture, the input variables are the temperature, rainfall and the present number of dengue cases. Windows of size 20 (ie the past 20 weeks' values of the input variables) are used to predict the number of dengue cases in 1, 8 or 16 weeks in the future.

The required datasets are provided in the `~/caffe/data/dengue` directory. As before, there is the `data` dataset to be used as input and the `label` dataset which are the list of actual dengue case counts to be compared with the model

predictions. The dimensions of the train and test datasets are

- train.h5
  - data: (723 3 20)
  - label: (723 3)

- test.h5
  - data: (181 3 20)
  - label: (181 3)

The first axis of all the datasets are the time-steps. Train and test datasets are split in a 80:20 ratio, corresponding to 723 and 181 time-steps respectively.

Each time-step of the `data` dataset contains 3 variables (dengue cases, mean temperature and mean rainfall, in that order) with a window size of 20. This accounts for the size of the second and third axes of the dataset.

For the `label` dataset, the second axis stores the T+1, T+8 and T+16 actual values in that order. Hence a slice layer is needed to choose one of the 3 labels to use as the prediction target.

Experiment
Try to do 1, 8 and 16 week predictions. Use a simple 1 to 3 layer NN structure and apply the techniques and lessons learnt from the previous labs. Be systematic in your experimentation and testing of your hypotheses. Apply your intuition to come up with different approaches, so that you do not just blindly try every possible combination of parameters.

You will need to summarize your findings and be prepared to present them (3-5 mins) tomorrow, with a 5min Q&A. You are encouraged to work together as a group.